John von Neumann Institute for Computing

”Second-generation” Skeleton Systems

M. Danelutto

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata
(Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 803-810, 2006.

http://www.fz-juelich.de/nic-series/volume33

"Second-generation" skeleton systems

M. Danelutto[a]

[a]Dept. Computer Science – University of Pisa – Largo Pontecorvo 3 – 56127 Pisa – Italy

Algorithmical skeletons, as originally introduced in late '80s, evolved under the pressure of users, on the one side, and designers, on the other side. The formers asking for new features, the latter perceiving the limits of the skeleton approach to parallel programming and trying to overcome them. In this work, we discuss the features that have to be tackled in a "second-generation", mature skeleton system. Actually, we propose to extend the requirements stated in previous work by Cole. We outline how these features have been taken into account in two programming environments we are currently developing at the University of Pisa, and we make a synthetic comparison with other known skeleton environments.
Keywords: structured parallel programming, skeletons, macro data flow, coordination languages, adaptivity, heterogeneity.

1. Introduction

When algorithmical skeletons were first introduced in late '80 [13] the idea had an almost immediate success. Several research groups started research tracks on the subject and come up with different programming environments supporting algorithmical skeletons. Darlington's group first developed functional language embeddings of the skeletons [18] and then moved to FORTRAN [19]. Our group designed P3L, which is basically a sort of skeleton parallel C [16,7]. Kuchen started the work on Skil [10] and eventually produced a C++ Skeleton Library [22]. Serot designed Skipper [24,25], which exploits the macro data flow implementation model introduced in [14]. The original definition of skeleton programming environment given by Cole in his book [11] "*The new system presents the user with a selection of independent "algorithmic skeleton", each of which describes the structure of a particular style of algorithm, in the way in which "higher order functions" represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of the appropriate skeleton.*" was almost completely embraced by these groups. In particular, the scientific community accepted the idea of a fixed selection of independent skeletons. All the above-mentioned skeleton systems developed in the '90 only provide the programmer with a fixed set of skeletons.

The fixed, immutable skeleton set was in the meanwhile a source of power and of a source of weakness for the skeleton systems. It allowed efficient implementations to be developed but also did not allow programmers to express neither non standard parallelism exploitation patterns nor patterns even slightly different from the ones provided by the supplied skeletons. A partial solution to the unavailability of skeletons modeling specific parallel patterns came from the implementation of skeletons as libraries, whose mechanisms adopted to exploit parallelism was partially known, such as the ones discussed in [17] or [9]. In the former case, skeletons are provided as plain C function calls. The input data stream and the output data stream are implemented by plain Unix file descriptors that are accessible to the user. Therefore the programmer can program his own parallel patterns and make them interact with the predefined skeletons just writing/reading data to/from standard file (pipe,

[0]This work has been partially supported by Italian national FIRB project no. RBNE01KNFP *GRID.it* and by the Italian national strategic project *legge 449/97* No. 02.00640.ST97.

actually) descriptors. In the latter case, skeletons are provided as collective MPI operations. The programmer can access the MPI communicator executing the single parallel activity of the skeleton (e.g. one pipeline stage or a task farm worker) and can freely manage the processors allocated to the communicator. Explicit primitives allow the programmer to receive tasks from the skeleton input stream and to deliver results to the skeleton output stream. In both cases, a limited degree of freedom is left to the programmer to program his own parallelism exploitation patterns either outside or inside the ones modeled by skeletons. Despite being around since long time and despite the progress made in skeleton system design and implementation, the skeleton systems did not take off as expected. Nowadays, the skeleton system usage is actually restricted to small communities grown around the teams that actually develop the skeleton systems.

Cole focused very well the problem in his *manifesto* [12]. Here he states that four problems have to be taken into account and solved to allow skeletons to gain significant popularity: ❶ *'propagate the concept with minimal conceptual disruption"*, that is skeletons must be provided within existing programming environments without actually requiring the programmers to learn entirely new programming languages ❷ *"integrate ad-hoc parallelism"*, i.e. allow programmers to express parallel patterns not captured by the available skeleton set ❸ *"accommodate diversity"*, that is provide mechanisms to specialize skeletons, in all those cases where specialization does not radically change the nature of the skeleton, and consequently the nature of the implementation, and ❹ *"show the payback"*, i.e. demonstrate that the effort required to adopt a skeleton systems is immediately rewarded by some kind of concrete results: shorter design and implementation time of applications, increased efficiency, increased machine independence of the application code, etc. While the second and the third points are more specifically technical, the first and the last one are actually more "advertising oriented". All these points, however, have impacts on both the way the skeleton systems are designed and on the way they are implemented.

In addition, we also claim that another small set of problems have to be tackled: ❺ *support code reuse*, that is allow programmers to reuse with minimal effort existing sequential code ❻ *handle target architecture heterogeneity*, i.e. implement skeletons in such a way skeleton programs can be run on clusters/networks/grids hosting heterogeneous computing resources (different processors, different operating systems, different memory/disk configurations, etc.) ❼ *handle dynamicity*, i.e. implement in the skeleton support proper support to handle typical dynamic situations, such as those arising when non dedicated processing elements are used (e.g. peaks of load that impair load balancing strategies) or from sudden unavailability of processing elements (e.g. network faults, node reboot). The first point comes from our P3L experience. P3L [7], and its "industrial" successor SkIE [8], both allowed portions of sequential code written in C, C++ and FORTRAN 77 to be included in skeletons. SkIE also allowed High Performance Fortran to be used in the building blocks of skeletons (e.g. in pipeline stages or in task farm workers). Users greatly appreciated this feature that allows to wrap existing code with minor effort and to reuse all the huge, existing library of (optimized) sequential code. The second and third points actually come from our experience in grid programming systems. Within the GRID.it FIRB three year Italian national project [21], we developed a structured parallel programming environment targeting clusters, networks and grids and based on the skeleton programming methodology: ASSIST [29,1,28]. Grid systems are dynamic and heterogeneous by definition [20] and any programming environment targeting grid architectures must include proper techniques and algorithms to take care of these two important aspects, possibly in an automatic and transparent way [26]. Overall, the solutions to the set of problems stated above should be considered the basis of "second generation" skeleton systems. In other words, mature skeleton technology should efficiently address all these problems. In this perspective, here we want

```
import muskel.*;

public static void main(String [] args) {
   ...
  Compute stage1 = new Farm(new doSweep());             // first stage, process the input param set
  Compute stage2 = new PostProcess();                   // second stage: sequential postprocess
  Compute mainProgram = new Pipeline(stage1,stage2);    // main program is a two stage pipeline
  ParDegree parDegree = new ParDegree(5);               // required parallelism degree
  ApplicationManager manager =
                  new ApplicationManager(mainProgram); // instantiate the application manager
  manager.setContract(parDegree);                       // now set the performance contract
  manager.inputStream("input.dat");                     // tell were are input task data
  manager.outputStream("output.dat");                   // tell were results must be stored
  manager.eval();                                       // compute in parallel
  ...                                                   // any code processing the results here ...
}
```

Figure 1. Sample **muskel** code

to discuss two "second-generation" experiences of our group in Pisa, namely the **muskel** one [15] and the ASSIST one [29,1,28]. Both are programming environments based on the skeleton structured parallel programming concepts. The former being a plain Java library exploiting macro data flow implementation techniques [14] derived from these used in Lithium [5], the latter defining a new programming language, which is actually a coordination language that uses skeletons to model parallelism exploitation patterns. ASSIST exploits implementation techniques derived from the implementation template methodology developed in P3L and adopted by other skeleton frameworks [22].

2. muskel

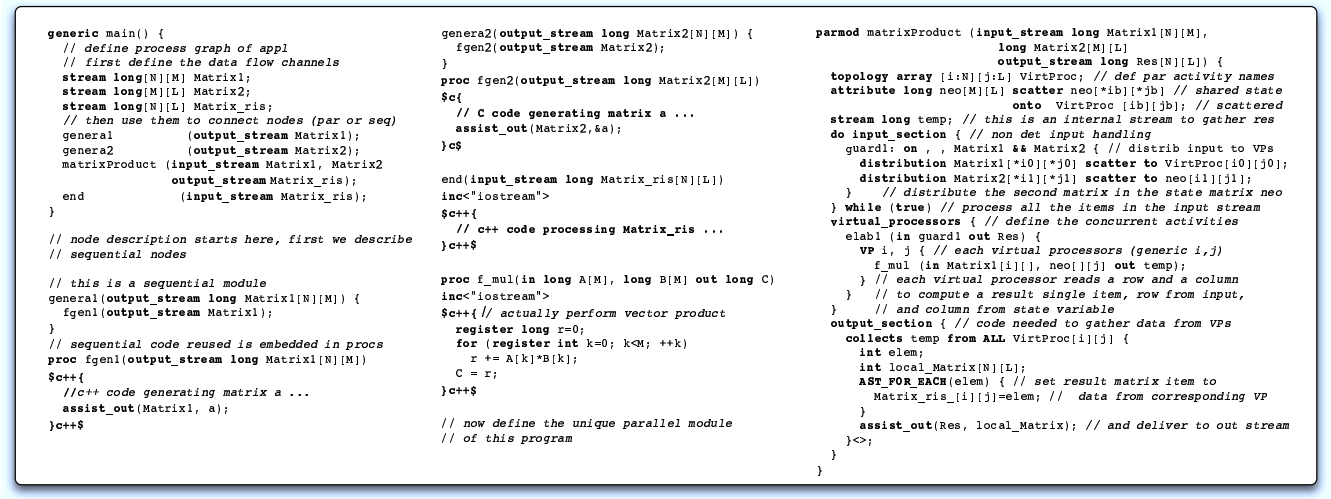
muskel (the name comes from the transliteration of μ-**skel**etons[1]) [15] is a full Java skeleton library providing user with usual stream parallel skeleton (pipelines, farms and arbitrary composition of farm and pipes). It is a compact, optimized subset of Lithium [5] and it has mainly being thought as a handy test bed to experiment new implementation strategies. Parallelism is exploited in **muskel** using plain java RMI. Remote interpreter objects are placed once and for all on all the processing elements possibly participating in the parallel computation. The skeleton library provides to automatically discover the processing elements where the remote interpreters have been placed and to recruit a suitable number of remote interpreters to schedule computations. Skeletons are implemented in **muskel** using macro data flow technology [14]: the skeleton program is translated into a data flow instruction graph. Instructions are fired when all the input tokens are available. A fireable instruction is simply scheduled for the execution on one of the available remote interpreters using RMI. Data flow instructions are actually "macro" data flow instructions. The user provides instruction functions as a parameter of the associate skeleton. In particular, sequential code to be used in these parameters is supplied as a `Compute` object, i.e. an object with an `Object compute(Object task)` method that returns a result after computing some sequential function on the input task object. In **muskel**, the skeleton program executed is not actually the one provided by the user: the skeleton program is first transformed to obtain its normal form as defined in [4] and then this normal form skeleton program is actually executed. A typical Java program using **muskel**

[1] we actually discovered recently that the Skeleton Library developed by Kuchen is called "muskel" on its web site. Here however, we use **muskel** to refer the Java library developed in Pisa and we refer to the other one as "Kuchen's Skeleton Library", to avoid name clashes

looks like the one in Figure 1. This program computes a two-stage pipeline, where the first stage is parallel (a task farm) and the second one is sequential. First the structure of the skeleton program is given. Then an `ApplicationManager` is instantiated and the performance contract (the parallelism degree, in this case) is passed to this manager along with the input and output files/streams. Eventually a single call is issued, the `manager.eval()` one, and this call takes care of all the steps needed to compute the skeleton program onto the stream of input tasks producing a stream of output results. In particular, the skeleton code is normalized and transformed into a macro data flow instruction graph, a discovery process is started and a number of remote interpreters congruent with the user supplied performance contract is contacted. The macro data flow instructions deriving from the skeleton code are staged to the remote interpreters. A thread is forked for each one of the remote interpreters recruited. The thread looks for fireable instructions in a *task pool* repository, delivers them to the associate remote interpreter and waits for the results of the computation. When results come back from the remote interpreter they are either delivered to the *result pool*, i.e. the place when they are taken to be delivered to the output stream, or reinserted in the proper target macro data flow instruction in the task pool. This macro data flow instruction can possibly become fireable. Immediately after the thread starts trying to fetch a new fireable instruction. In case a remote interpreter becomes unavailable (e.g. due to a network or to a node failure) the **`muskel`** application manager arranges to recruit a new one, if available. The task(s) left un-computed by the missing interpreter are put back to the task pool and they will be eventually re-scheduled to a different interpreter. Results achieved with **`muskel`** are very good both in terms of load balancing and in terms of fault tolerance and absolute performance/efficiency [15]. A minimal effort is required to experienced Java programmers to use **`muskel`**: basically, a small effort to implement the `Compute` interface in the existing application dependent code, plus the launch of the remote interpreter RMI objects on the available processing elements (both plain RMI and `rmid` versions of the remote interpreter are available) (thus addressing problem ❶). `FileInputStream` and `FileOutputStream` objects are passed to the manager to provide input task and retrieve output results. Therefore specialized parallel patterns can be programmed that interact with the existing skeleton (programs) via the streams, in the flavor of what happened in [17] with Unix file handles. Furthermore, recent improvements in the library [27] allow programmer to interact directly with the task/result pools. In particular, the user can fetch results from the result pool and can use them to build new (possibly fireable) macro data flow instructions to be inserted in the task pool. With such mechanisms, the programmer can either program his own macro data flow graphs or even implement completely new skeletons and add such skeletons to the library. Overall these two aspects allow both to integrate ad-hoc parallelism and to accommodate diversity (❷❸). The pay-back offered by **`muskel`** is shown by Figure 1, clearly evidencing the negligible amount of code needed to get a fully working, efficient parallel application (❹). Target architecture heterogeneity is handled naturally in **`muskel`** due to portability features of the JVM and RMI (❺). Dynamicity is handled in the `ApplicationManager`, where fault tolerance is also dealt with (❻). The only problem not actually solved is the support to code reuse (❼), as only Java code can be easily reused. The structure used to exploit parallelism heavily relies on serializability of code and it will be difficult to adapt to support C, FORTRAN or even C++ code reuse.

3. ASSIST

ASSIST (A Software development System based on Integrated Skeleton Technology) is a programming environment aimed at supporting parallel/distributed application development on clusters and networks of workstations as well as on grids. The environment implements the ASSIST co-



```
generic main() {
  // define process graph of appl
  // first define the data flow channels
  stream long[N][M] Matrix1;
  stream long[M][L] Matrix2;
  stream long[N][L] Matrix_ris;
  // then use them to connect nodes (par or seq)
  genera1          (output_stream Matrix1);
  genera2          (output_stream Matrix2);
  matrixProduct (input_stream Matrix1, Matrix2
                 output_stream Matrix_ris);
  end             (input_stream Matrix_ris);
}

// node description starts here, first we describe
// sequential nodes

// this is a sequential module
genera1(output_stream long Matrix1[N][M]) {
  fgen1(output_stream Matrix1);
}
// sequential code reused is embedded in procs
proc fgen1(output_stream long Matrix1[N][M])
$c++{
  //c++ code generating matrix a ...
  assist_out(Matrix1, a);
}c++$
```

```
genera2(output_stream long Matrix2[N][M]) {
  fgen2(output_stream Matrix2);
}
proc fgen2(output_stream long Matrix2[M][L])
$c{
  // C code generating matrix a ...
  assist_out(Matrix2,&a);
}c$

end(input_stream long Matrix_ris[N][L])
inc<"iostream">
$c++{
  // c++ code processing Matrix_ris ...
}c++$

proc f_mul(in long A[M], long B[M] out long C)
inc<"iostream">
$c++{// actually perform vector product
  register long r=0;
  for (register int k=0; k<M; ++k)
    r += A[k]*B[k];
  C = r;
}c++$

// now define the unique parallel module
// of this program
```

```
parmod matrixProduct (input_stream long Matrix1[N][M],
                      long Matrix2[M][L]
                      output_stream long Res[N][L]) {
  topology array [i:N][j:L] VirtProc; // def par activity names
  attribute long neo[M][L] scatter neo[*ib][*jb] // shared state
                           onto  VirtProc [ib][jb]; // scattered
  stream long temp; // this is an internal stream to gather res
  do input_section { // non det input handling
    guard1: on , , Matrix1 && Matrix2 { // distrib input to VPs
      distribution Matrix1[*i0][*j0] scatter to VirtProc[i0][j0];
      distribution Matrix2[*i1][*j1] scatter to neo[i1][j1];
    }    // distribute the second matrix in the state matrix neo
  } while (true) // process all the items in the input stream
  virtual_processors { // define the concurrent activities
    elab1 (in guard1 out Res) {
      VP i, j { // each virtual processors (generic i,j)
        f_mul (in Matrix1[i][], neo[][j] out temp);
      } // each virtual processor reads a row and a column
    }   // to compute a result single item, row from input,
  }     // and column from state variable
  output_section { // code needed to gather data from VPs
    collects temp from ALL VirtProc[i][j] {
      int elem;
      int local_Matrix[N][L];
      AST_FOR_EACH(elem) { // set result matrix item to
        Matrix_ris_[i][j]=elem; //  data from corresponding VP
      }
      assist_out(Res, local_Matrix); // and deliver to out stream
    }<>;
  }
}
```

Figure 2. Sample ASSIST code

ordination language, which is actually a language that allows to express arbitrary process graph applications, were each node in the graph can either be sequential or parallel, and nodes communicate via data flow streams [29]. Parallel nodes can be expressed using the `parmod` skeleton. A parmod (a generic **par**allel **mod**ule) can have multiple input stream and output streams. Programmer can implement arbitrary non-deterministic control on the input streams as well as to generate an arbitrary number of output items on the output streams. A parmod defines a set of logically concurrent parallel activities. The keyword *logically* refers to the fact that the ASSIST compiler and run time completely take care of executing them on the set of available/required processing elements in a transparent and optimized way. Such logically parallel activities, referred to as *virtual processors* in the ASSIST jargon, can be named as multidimensional arrays or as 'anonymous". In the former cases, virtual processors are distinguished in the program by indexes: input data can be delivered to specific virtual processors, or they can be broadcasted/multicasted to the virtual processors (this is used to implement data parallel computations as well as very specific parallel skeletons/patterns). In the latter case, the parmod only specifies the number of parallel activities: all the parallel activities are equivalent and input data can only be delivered to a generic virtual processor for processing (this is exploited to implement task farms). State variables can be shared among the virtual processors. The owner computes rule holds in case the shared variables are scattered across the virtual processors: a vector state variable x scattered across a vector of virtual processors allows virtual processor i to read any value $x[j]$ but to write only the value $x[j]$. The code executed by virtual processors, as well as the code executed by the sequential nodes, can be specified using C, C++ and FORTRAN77, at the moment, and we have already experimented the possibility of using Java code too. Virtual processor computation is triggered by the availability of all the input data specified in the virtual processors code. Therefore data flow execution mode is assumed. Virtual processors activities can be iterated and the compiler and run time support provide to insert proper synchronization to avoid processing data relative to different iterations. Figure 2 shows an ASSIST program with two sequential processes generating each a stream of matrixes and a parmod node multiplying such matrixes exploiting data parallelism. The ASSIST parmod represents the major innovation of ASSIST with respect to previous skeleton systems developed at our group. The parmod represents a *generic* parallel module. By specializing this generic construct many par-

PROBLEMS	NOTABLE SOLUTIONS IN: *muskel*	*ASSIST*	*eSkel*	*Kuchen's Skelton Library*
minimal conceptual disruption ❶	plain Java lib		plain MPI collectives	plain C++ lib
ad-hoc parallelism ❷	macro data flow level accessible to user + access to streams	parametric parmod	protected MPI communicators for parallel skeleton building blocks	variety of combination of (data parallel) skeletons
accommodating diversity ❸	same as above	same as above	parametric skeleton calls	
show pay-back ❹	OO lib expressive power, fast application development	fairly fast application development, highly efficient object code	fast application development	OO lib expressive power, fast application development
code reuse ❺	Java	C C++ FORTRAN	C C++	C C++
heterogenity ❻	guaranteed by Java	compiler + run time	guaranteed by MPI †	guaranteed by MPI †
dynamicity ❼	application manager	module + application manager		

† data type handling is in charge to the programmer, i.e. he should not use plain `MPI_Byte` data messages.

Figure 3. Summary of solutions to problems ❶–❼ in several advanced skeleton environments

allel skeletons can be derived: pipelines and farms, map/forall and reduce with or without shared state. The implementation of parmod is highly optimized and both relies on a huge compilation process and on an optimized run time system the ASSISTlib. The price to pay is a somehow heavy language. Also, the possibility to express arbitrary graphs of parallel/sequential nodes is a notable step away from previous experiences. As **`muskel`**, ASSIST supports autonomic control of parallel modules and of the overall application [6]. A parmod can be executed in such a way that the user asks a given performance contract to be satisfied. In this case, the parmod automatically provides to keep the contract satisfied, if possible: exploiting the knowledge coming from the parmod analytic performance models the parmod control dynamically adapts the number of resource used to execute the parmod, in such a way the user supplied performance model is satisfied. Some partners of the national project GRID.it including CNR, the Italian National Research Council, and ASI, the Italian Space Agency, currently use ASSIST to develop different kinds of applications: graphics (isosurface detection), bioinformatics (protein folding), "social" applications (sea oil spill detection, landslip detection) and chemistry (ab-initio molecule simulation). The major ASSIST pay-back is given by the results achieved when implementing these complex, possibly multidisciplinary applications: the development time was drastically reduced with respect to the time needed to develop equivalent, traditional parallel applications (e.g. applications programmed using MPI) and the efficiency achieved is almost the ideal one, in most cases (❹). However, the need to learn a completely new and fairly untraditional parallel language does not help to propagate the skeleton concept with minimal conceptual disruption (❶). Ad hoc parallelism is de facto integrated, through the noticeable reconfigurability of parmod (❷). The same feature allows accommodating diversity upon specific programmer/applications needs (❸). Code reuse is supported (C, C++, FORTRAN code reuse is already supported and Java support is forthcoming, ❺), heterogeneity is handled (current ASSIST version produces code running on networks of mixed Linux/Intel and MacOSX/PowerPC machines, ❻) and dynamicity is handled via the module and application managers implementing autonomic QoS control (❼). Actually, there are much more interesting properties and features in ASSIST, that are not being considered here because not relevant to the second-generation skeleton discussion. ASSIST supports its own component model, as an example [2], and it also supports seamlessly interaction with both CORBA/CCM and Web Services world. The interested reader may refer to [3].

4. Conclusions

We discussed the solutions given to problems ❶–❼ stated in Sec. 1 by two skeleton programming environments currently being developed in Pisa. Figure 3 shows a comparison of notable solutions given by either these two environments or by eSkel and Kuchen's C++ skeleton library. This summary table points out two aspects. Some environments react to problem ❷ and ❸ by providing limited/controlled access to the implementation level, in such a way users can program their own parallelism exploitation patterns. eSkel does it by allowing users to program parallel activities within the single component of a skeleton (e.g. a pipe stage), while `muskel` allows to program parallel activities outside the skeletons but interacting with the skeletons (we are currently working to extend these `muskel` features, indeed). A solution to problems ❷ and ❸ should probably provide both these possibilities. As skeleton systems are more and more oriented to give the user the possibility of programming his own skeletons, solutions such as the one adopted in [23] to guarantee controlled accesses to the implementation framework are needed. The other aspect to consider is that a tradeoff has to be found between the number of parameters needed to specify a skeleton and the skeleton system expressive power. ASSIST provides a highly customizable parmod skeleton, but the learning curve needed to make an efficient use of it is not negligible. Other systems provide much more strict skeletons, but they also must release the constrain to leave the implementation layer invisible to users, in order to guarantee solutions to ❷ and ❸. Library implementation of skeleton systems seems to guarantee a better framework to support this idea than implementations providing a full programming language. However, some compile time solutions that demonstrated very efficient in the implementation of ASSIST, as an example, look like very hard to implement in a library. Therefore techniques combining some kind of just-in-time compiling with library skeleton implementation should probably be exploited. Last but not least, solutions to problem ❺ to ❼ are fundamental to the success of skeletons systems as they guarantee to preserve the investements made in sequential software development and to target a larger and more significant class of architectures. Both `muskel` and ASSIST experience demonstrated that the ability of adapting application execution to varying features of an heterogeneous target architecture is a key point in convincing a user to migrate to a skeleton programming environment.

References

[1] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzoloand M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In *Proc. of Intl. Conference EuroPar2003: Parallel and Distributed Computing*, number 2790 in LNCS. Springer, 2003.

[2] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for High-Performance Grid Programming in GRID.it. In *Component modes and systems for Grid applications*, CoreGRID. Springer, 2005.

[3] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured Implementation of Component based GRID programming environments. In *FGG: Future Generation Grids*, CoreGRID. Springer Verlag, 2005.

[4] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimisations. In *Proc. of the IASTED International Conference Parallel and Distributed Computing and Systems*, pages 955–962. IASTED/ACTA Press, November 1999. Boston, USA.

[5] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. Elsevier Science.

[6] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dy-

namic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, LNCS. Springer Verlag, 2005. to appear.

[7] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Conc. Practice and Experience*, 7(3):225–255, 1995.

[8] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, December 1999.

[9] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In *Proceedings of Euro-Par 2005: Parallel Processing*, LNCS. Springer Verlag, 2005. to appear.

[10] G. H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196(1–2):71–107, April 1998.

[11] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[12] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[13] M. I. Cole. A "Skeletal" Approach to Exploitation of Parallelism. In C. R. Jesshope and K. D. Reinartz, editors, *CONPAR 88*, British Computer Society Workshop Series. Cambridge University Press, 1989.

[14] M. Danelutto. Dynamic Run Time Support for Skeletons. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and H. J. Sips, editors, *Proceedings of the International Conference ParCo99*, volume Parallel Computing Fundamentals & Applications, pages 460–467. Imperial College Press, 1999.

[15] M. Danelutto. QoS in parallel programming through application managers. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based processing*. IEEE, 2005. Lugano.

[16] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and support of massively parallel programs. *FGCS*, 8(1–3):205–220, July 1992.

[17] M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, LNCS, No. 1900, pages 1175–1184. Springer Verlag, August/September 2000.

[18] J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel Programming Using Skeleton Functions. In M. Reeve A. Bode and G. Wolf, editors, *PARLE'93 Parallel Architectures and Langauges Europe*. Springer Verlag, June 1993. LNCS No. 694.

[19] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler. Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages. In *Third Parallel Computing Workshop (PCW'94)*. Fujitsu Laboratories Ltd., November 1994.

[20] I. Foster and C. Kesselman (Editors). *The Grid 2 Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, December 2003.

[21] Grid.it community . The GRID.it home page, 2005. http://www.grid.it.

[22] H. Kuchen. A Skeleton Library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. "Springer" Verlag, August 2002.

[23] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From Patterns to Frameworks to Parallel Programs. *Parallel Computing*, 28(12):1663–1683, 2002.

[24] J. Sérot. Tagged-token data-flow for skeletons. *Parallel Processing Letters*, 11(4):377–392, Dec 2001.

[25] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the SKiPPER project. *Parallel Computing*, 28(12):1785–1808, Dec 2002.

[26] D. Snelling and K-Jeffrey et al. Next Generation Grids 2 – Requirements and Options for European Grids Research 2005-2010 and Beyond, 2004. ftp://ftp.cordis.lu/pub/ist/docs/ngg2 eg final.pdf.

[27] S. Susini. Introduction of new features in a Java parallel programming environment *in italian*, June 2005. Final stage report of the "Diploma in Informatica", Dept. Computer Science, Univ. of Pisa.

[28] The ASSIST team. ASSIST home page, 2005. http://www.di.unipi.it/groups/architetture/Assist.html.

[29] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Computing*, 12, December 2002.